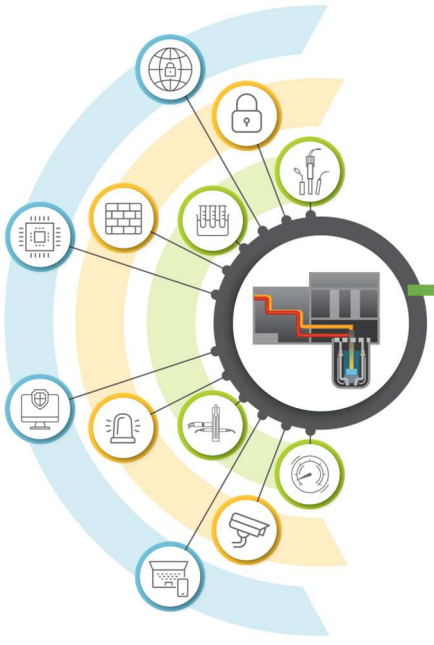


Model-based Hierarchical Reinforcement Learning for Improved Physical Security Design: A Protoype

Prepared for
US Department of Energy

Nathan Shoman

September 2025

SAND2025-11642R

Acknowledgments

This work was funded through the U.S. Department of Energy's Office of Nuclear Energy (DOE-NE), Advanced Reactor Safeguards and Security (ARSS) campaign. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The author would like to recognize the contributions by Alan Evans, Austin Orr (physical security expertise and supporting documentation), and Kannad Khanna (A* pathing algorithm implementation).

The authors would also like to acknowledge the open source contributions of Eclectic-Sheep/sheeprl, Jormeli, and danijar for their work. Algorithms implemented in this work were strongly influenced by their open source code.

This page intentionally left blank.

CONTENTS

This page intentionally left blank.

LIST OF FIGURES

This page intentionally left blank.

EXECUTIVE SUMMARY

Prior work in FY24 developed an adversarial AI agent aid in path analysis of physical protection systems. This agent, trained using a model-based reinforcement learning algorithm, was able to successfully learn the most vulnerable path in facilities. It was able to extend the current state of practice for physical protection design by exhibiting dynamic behavior based on current environmental conditions. Whereas PathTrace largely performs a static, graph-based analysis, the AI agent was able to make decisions based on relative position in the facility, current conditions (was the adversarial agnet discovered?), and proximity to secondary targets. The agent demonstrated some novel capabilities, but had limitations that need to be resolved before it can be used for production purposes. For example, the adversarial agent generalizes poorly and takes a relatively long time to train. Nonetheless, there is still considerable promise for developing the adversarial agent further in order to explore even richer, more dynamic behaviors (e.g., adversary motivations, environmental debris, and more).

This work considers a complementary idea; development of a planning agent. The planning agent is envisioned as an auto-complete-like tool that can help accelerate security system design by human experts. The agent would respect existing barriers and sensors placed by a human expert while offering cost-effective suggestions (i.e., implicitly balancing effectiveness with cost) to improve the design. The goal is for this agent to be part of an expert's toolbox, not to totally upend the current state-of-practice, or to displace human experts. The ultimate goal would be concurrent training of both the adversarial and planning agent together, to learn entirely through self-play. This would represent an entirely new way of performing system deign.

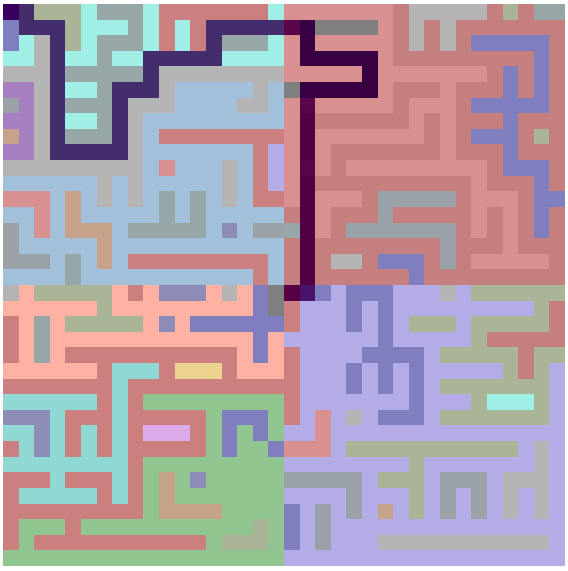
We selected a hierarchical, model-based reinforcement learning algorithm to serve as the planning agent. This is an extension of concepts used in the prior FY24 adversarial agent work. There, we had a single agent acting an environment. Here, we have two different sub-agents (policies), working together, to form a complete agent. There is a manager policy, which can select abstract goals on slower time scales, and a worker, which performs primitive actions to reach goals selected by the manager.

It is worth noting that this class of algorithm is challenging to work with. From our understanding, our work is one of the first successful uses of model-based reinforcement learning (MBRL) in nuclear energy[1], and likely the first hierarchical model-based reinforcement learning application in nuclear energy. Further, this work is one of the first known attempts to apply AI to perform a design tasks in nuclear energy. Consequently, there were significant implementation challenges and the bulk of the work was focused on successful implementation and algorithm design. The results presented here are very low technology readiness level as a consequence of the lack of related literature, but still represent a significant step forward in the pursuit of applied AI for design.

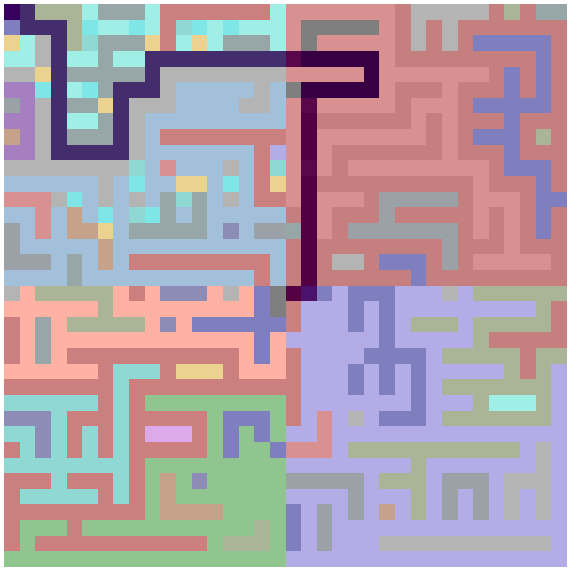
[1]We only found one other example of MBRL in nuclear energy and it was a PhD dissertation using MBRL for accelerator beam control

We trained a hierarchical, model-based reinforcement learning algorithm to perform auto-complete-like design of a physical protection system. We had the agent successfully learn high impact edits, as seen in Figure 0-1 below. This initial work used a reduced scale version of the real problem, but with a full rule set (i.e., we made no compromises in terms of agent capabilities, only reduced problem size). In Figure 0-1 below, each color represents a different PPS element, each with it's own probability of detection, tool-based delay, and approximate cost (in dollars). We combine all these metrics together to create a measurement of path length that can be used to train the agent to improve the design of a layout.

We create a randomized environment for the planning agent to learn how to improve designs, entirely through self-play. It's not feasible to develop limitless real designs, so we trained an autoencoder on different maze design strategies. Then, this autoencoder is used to generate initial layouts for the planner agent. We then randomly assign sensor and barriers to different areas of the layout. This finalized maze layout with colored PPS elements and corresponding ruleset is used to train the planning agent.



(a) Randomly generated layout before edits have been made.



(b) Randomly generated layout after edits have been made. Note that both the path itself has changed as has the cost of the path.

Figure 0-1.: Maze after agents has made edits. Note that edits are concentrated in the top left patch, whereas the others have been unedited. The highlighted path length (shortest path from a fixed origin in the top left) has changed, as has the entire global path lengths and top left patch path length.

We found that the trained agent learned how to make strategic edits, with a limited budget of edits while balancing cost (dollars), but multiple challenges still remain. The most notable remaining challenges at the prototype stage is the low training stability (some runs learn effective strategies, others don't) and the stationarity of the agent (the agent tends to only edit a subset of the entire design area). We have several hypotheses for these limitations, which can be considered in out year work. Although there some challenges remain, this work represents a significant step forward in applied AI for nuclear.

Nomenclature

AI Artifical Intelligence

CNN Convolutional Neural Network

DEPO Design and Evaluation of Physical Protection Systems

HRL Hierarchical Reinforcement Learning

MBRL Model-Based Reinforcement Learning

MDP Markov Decision Process

ML Machine Learning

PCGRL Procedural Content Generation via Reinforcement Learning

PPS Physical Protection System

RL Reinforcement Learning

RSSM Recurrent State Space Model

RGB Red Green Blue

VAE Variational Autoencoder

This page intentionally left blank.

1. INTRODUCTION

Physical security is an important component of nuclear facilities. Any applicant seeking to build and operate a nuclear fuel cycle or nuclear power facility is required to demonstrate the performance of their proposed system. Designing an effective system that is also cost effective can be challenging, particularly for advanced nuclear power facilities where operating margins may be smaller. Consequently, it is beneficial to consider physical security during the design stage in order to avoid costly future retrofits needed to remedy initial deficiencies.

There have been some recent advances in contemporary physical security design to try to remedy these challenges. For example, recent work by Evans [2] on security for small modular reactors highlights the need to consider small details; everything from where workers will live to unique sabotage targets of advanced reactors, to fully optimize cost. However, considering all these impacts and searching the space of possible physical security solutions can be time consuming an expensive. The current state of practice is based around the iterative DEPO methodology (Design and Evaluation Process Outline) [3]. This process involves first defining performance requirements and then iterating over several steps:

- Evaluating safety and operational considerations
- Performing PPS design
  - Delay, Detection, and Response
- Evaluating design
  - Path Analysis
  - Force-on-Force Analysis

This methodology has been effective historically, but is slow, expensive, and subject to bias. Human expertise is used at every step which naturally limits the scope of possible scenarios considered. There is no guarantee that a system is "optimal" globally as designs are evaluated against regulatory targets within a finite historical and cultural context that functions as priors on the final design. This work aims to improve the search of possible solutions through the use of artificial intelligence (AI).

Specifically, this work aims to develop a *planning* agent to complement the previously developed adversarial agent in FY24. The planning agent is envisioned to act as an "auto-complete" tool of sorts to help accelerate design by providing human experts with new tools. This planning agent will suggest a layout of physical security elements that would complement existing user specified elements. The layout will be informed by user-specified options, such as the trade off between footprint and cost. We emphasize that the planning agent is designed to complement a human expert during one part of the DEPO methodology (Path Analysis), not to replace human expertise or the DEPO methodology.

The planning agent represents state-of-the-art research in both physical security and machine learning. To the best of our knowledge, this is one of, if not the first, application of hierarchical model-based reinforcement learning and the second application [1] of flat (non-hierarchical) model-based reinforcement learning in nuclear energy.

While groundbreaking, the work presented in this report is not a final product, but represents progress in developing a prototype planning agent. The adversarial agent's goal is straightforward; reach one (or more) sabotage targets. However, the planner agent goal is much more complex. The planner must design a (potentially) large scale physical protection system (PPS) with a complex objective that balances probability of detection, delay, cost, and physical footprint. The work here is a scaled down version of the final planner agent, but still retains the full game rules and dynamics. Our successful proof-of-concept here demonstrates the feasibility of AI to help accelerate system design in the nuclear energy domain.

[1] The first is believed to be our prior work on the adversarial agent from FY24.

2. RELATED WORK

The current state of practice for physical security follows the steps outlined in DEPO [3]. Execution of each step requires one or more analytical tools. This work focuses on the *path analysis* part of the DEPO process. Specifically, the goal is to accelerate the design of physical protection systems in a path analysis context by providing suggestions to human experts. The current state of practice for this step is to perform manual design using the PathTrace tool. PathTrace allows users to graphically place different security elements (e.g., walls, doors, sensors, etc.) on a scaled facility image. Then, under the hood, PathTrace uses Dijkstra's algorithm to calculate the most vulnerable path by representing physical security elements as a weighted graph. This approach does find the "most vulnerable path", but only in a static sense. It cannot perform dynamic analyses based on environmental conditions, which we have attempted to address in related work on reinforcement learning for adversarial behavior discovery. We refer readers to [4] for further details as the specifics of PathTrace will not be covered in depth here.

Flat, model-free reinforcement learning has been used for a variety of control and design optimization tasks in different domains. We will not enumerate all examples here, but a few applications include industrial control systems [5], robotic manipulation [6], parameter tuning for planning algorithms [7], and nuclear power plant operation optimization [8, 9]. Perhaps the most relevant related work to this project is Procedural Content Generation via Reinforcement Learning (PCGRL) [10]. In PCGRL, the authors trained a reinforcement learning algorithm as a video game level design agent. Specifically, an agent was trained to create viable video game levels from an initial random design. There were several key findings regarding action space and environmental representation that are key to this work.

This work focuses primarily on hierarchical model-based reinforcement learning, of which, there is significantly less work on model-based reinforcement learning, and heirarchical extensions, for nuclear. It's out understanding that this work is among the first in the nuclear energy space.

This page intentionally left blank.

3. PROBLEM DESCRIPTION

The goal of this work is to develop a "designer" or "planner" machine learning agent that can assist human experts in physical security system design. This agent will act as an auto-complete, with a human-in-the-loop and human-specified controls. The envisioned workflow is for a human to place some critical infrastructure/goals along with some initial security elements. Then, for a particular area of the layout, the designer agent could provide suggestions for security element placement based on user-specified criteria like a balance between cost, footprint, detection, and delay.

We cast this problem into the path analysis context of the DEPO methodology. Here, PPS are discretized into individual cells that represent some span of physical space. Each cell has an assigned object type that represents the presence or absence of a PPS element. For example, one cell might represent a wooden door where as another might represent a passive infrared sensor. The evaluation criteria is based on the path cost from any cell outside of the PPS design to some goal target(s). The cost of this path, in terms of difficulty for the adversary and design effectiveness, is based on the cumulative probability of detection and delay between the start point and goal.

Developing a designer agent is very challenging due to both the algorithmic complexities, environmental design, evaluation criteria, and reward specification. This work explored two different hierarchical reinforcement learning (HRL) algorithms to try to solve the problem. The first algorithm had several architectural limitations, so it was abandoned in favor of a more robust algorithm.

Reinforcement learning (RL) refers to a subset of machine learning where an agent [1] learns optimal behavior in an environment to solve a specified task. More formally, the goal is to train an optimal policy, $\pi_\theta(a_t|s_t)$, that maximizes expected returns over some temporal horizon. These polices are parameterized by θ and are often taken to be trainable neural networks.

A simple example might be training an agent to play classic Mario or teaching a robot to do a backflip. It is important to note that the reward function need not be differentiable. This allows for complex reward specifications that don't have analytical expressions (e.g., Mario collecting coins and reaching the flag while avoiding monsters). The general reinforcement learning loop is shown below in Figure 3-1.

This work focuses on hierarchical reinforcement learning, which extends the concept of reinforcement learning into a hierarchy of policies. Here, there might be a "manager" and "worker" agent wherein two different rewards are specified and both agents work together to solve a task. The advantage of having two separate agents is the ability to act on different temporal scales. Here, the manager might select a goal state for the worker to reach, without dictating how to do so. An example would be a manager

[1] We adopt the conventional nomenclature for agent. Some recent work will refer to large language models with tool access as "agents", but we use agent to refer to a policy-based AI construct acting inside an environment with a specified reward structure.

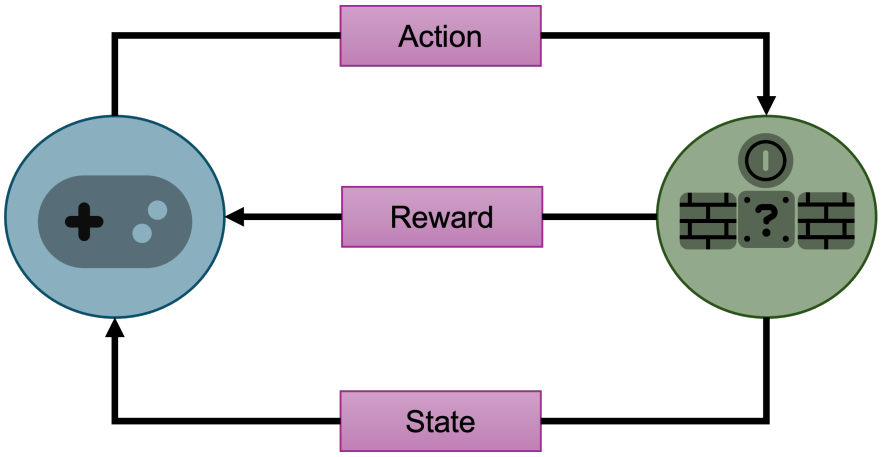


Figure 3-1.: Generalized reinforcement training loop

specifying the position of Mario on the screen while relying on the worker to execute the inputs to get Mario to that position.

Project Goal Summary

The goal of this work is to train a HRL agent to edit a PPS design according to user-defined trade-offs between various competing metrics like cost, delay, and footprint. The agent should be able to make performance aware edits to PPS elements. For example, the agent might swap a wooden door to a metal door. The goal here is to demonstrate a prototype agent, working in a scaled down environment, but otherwise using the full set of design capabilities.

Why reinforcement learning?

Reinforcement learning is notorious for being difficult to train and implement with hierarchical systems even more challenging, so the reader might ask why this approach was chosen. Reinforcement learning has commonly been studied in the context of video games as they fit well within a Markov Decision Process (MDP) framework. Path analysis can similarly be cast into a MDP framework fit for use within reinforcement learning. RL has several benefits that could translate to new insights for physical security; sequential decision making, real-time/dynamic learning, and abstract (non-differentiable) goal specification. Hierarchical reinforcement learning extends this further and adds the potential for skill composition, transfer learning, and more complex goal specification.

This work is centered on physical security, but could be extended to other problems and domains. We hope that this work represents just the first step for research of these techniques in the context of nuclear energy. There could be a future wherein learned policies could be composed into more complex systems for even more elaborate system designs. A compositional approach to applied AI could open the door to entirely new methods for designing and engineering systems.

4. METHODOLOGY: ALGORITHM

This work focuses on the application of HRL to designing a physical security system. HRL is an extension of RL where the multiple policies are organized into a hierarchy. For example, there might be high-level and low-level (and sometimes more) policy that jointly work towards some goal.

HRL was initially chosen as a technique precisely because of potential for a split goal representation. There is a natural need to conceptualize separate high and low-level goals for the physical security design problem. As a concrete example, we might want the manager (high-level policy) to specify more abstract goals like instructing the worker to construct a "funnel" between zone 1 and zone 2. This "funnel" goal would aim to funnel adversaries to different parts of the facility. The manager would not directly specify *how* to accomplish this, instead relying on the worker (low-level policy) to learn a reasonable action sequence for reaching the specified goal.

Other HRL concepts besides manager/worker architectures exist as well. For example, we might train a single high-level action policy that can execute skills or tools that are learned through exploration and self-play. The agent would then learn how to use these tools to accomplish tasks while dynamically swapping between them. These tools, in a physical security context, could act like paint brushes on a canvas. The high-level agent might swap between a spiral wall brush and a rectangular sensor brush.

This work considered two different HRL algorithms, each with a different approach for composing policies in a hierarchy. First Option-Critic [1] (similar to the paint brush example) and later Director [11] (the manager/worker paradigm example). Director is currently being used to develop the overall PPS planner agent due to algorithmic limitations and challenges encountered when using option-critic. Although option-critic wasn't successful in this application, we briefly discuss it's use for this problem and the failure modes that were encountered in order to share these experiences with the community.

4.1. Initial algorithm: Option-Critic

Option-critic was one of the earlier historical attempts to introduce temporal abstraction in to reinforcement learning. Generally, the concept of option-critic is to learn a set of internal policies (i.e., tools/skills or workers), a termination policy, and a policy over options (i.e., high-level tool user or manager). The idea is that the a set number of "options" would be initialized that a separate, higher-level policy could swap between to accomplish some task. The termination policy would be used to determine when a different option should be selected. An overview of the option-critic is shown in Figure 4-1 below. Rather than providing an exhaustive description and proof of the algorithm itself, we refer the readers to the original paper for further details [1].

Option-critic had a number of appealing qualities:

- Relatively straightforward implementation

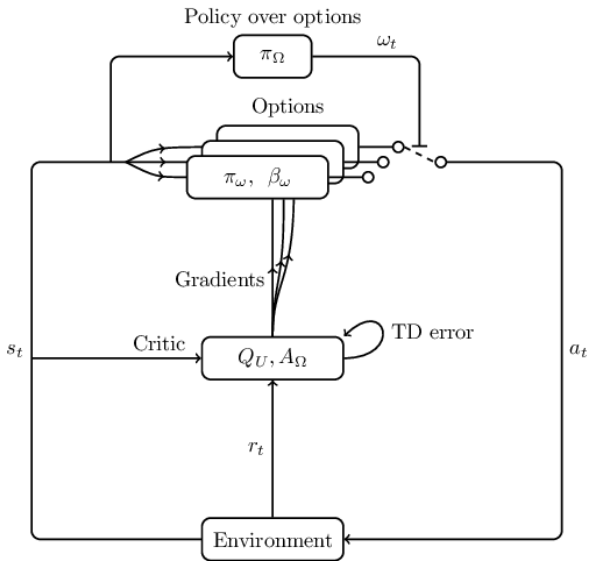


Figure 4-1.: Overview of the option-critic architecture [1]

- Intuitive architecture
  - Different options can be thought of as "brushes" to use on the area
- Separate manager (policy over options) and worker (options) observation spaces

The ability to separate the observation space for the policy over options was thought to be particularly important. This would allow the manager and worker to see different views and focus on different tasks. Specifically, the manager could see a global view of the world whereas the worker would be restricted to a local view.

For any HRL algorithm, there's many components that must be synchronized and coordinated to successfully learn to perform a task. The team spent several weeks implementing, testing, and debugging option-critic. However, ultimately the decision was made to swap to a different algorithm, Director, due to implementation challenges with option-critic. In hindsight, it's possible that there were experimental design issues that was causing option-critic to fail to learn, not necessarily the algorithm itself. Triaging these problems is non-trivial as it can be difficult to disentangle environmental specification errors and algorithmic implementation errors. Below are the series of failure modes encountered when implementing and deploying option-critic.

4.1.1. Premature option termination

The most significant challenge was the repeated early option termination. A termination gate (trainable neural network used to construct a Bernoulli distribution) evaluates, after each action in the environment, to terminate the current options. This enables option-critic to use different strategies for different lengths of time. For example, one option could be used for 50 steps whereas another might be only used for 5, and so on.

Our implementation had very short option terminations (i.e., options would only be executed for ≈ 5 steps or less before swapping), which creates many issues that leads to the algorithm not learning. Various mitigation strategies were incorporated, including warm-up gating (hard minimum option length during startup), termination bias scheduling (bias on the termination, β, before the final sigmoid activation to tilt termination probability), and minimum duration action masking (certain actions were masked and couldn't be selected until an option had been run for a minimum duration). We also considered an explicit penalty to option termination, but that could have had unseen consequences. For example, an option termination penalty would reshape advantages as the penalty would interact with the discounted returns. If the penalty is too big, it dominates the advantage, too small and it's washed out.

Ultimately, none of these were successful and diagnostic signals frequently conflicted with each other. It's not clear if the failure of these mechanisms to increase the option length was due to the mechanisms themselves or another undiagnosed error in the environmental setup (there were numerous fixes and adjustments to the environment made when developing Director).

4.1.2. Reward scaling

Reward scaling was quite challenging, even more so within option-critic (more details on rewards are outlined in Section 5.4). Algorithms learn best when rewards are smooth, continuous, and not peaky (i.e., large spikes are bad). Option-critic has two different rewards, one for the worker and one for the manager. The worker should see the raw reward distribution, so we opted to use simple scaling and clipping to try to constrain rewards to the $[-5, 5]$ interval. We used PopArt [12] on the manager, along with a scalar, to improve learning by keeping the reward distribution more stationary. Reward clipping was challenging to calibrate properly as clips that are too small could result in masked learning signals. Clips that are too large result in large gradients and peaky rewards that are difficult to optimize.

4.1.3. Miscellaneous mitigations

Several other improvements to option-critic were also added to stabilize learning, although none of them mitigated the continual option termination collapse. These included:

- TD target clamp during early stages of training
- Prioritized replay sampling based on TD-error
- RMS reward normalization
- Warmup learning for the policies before training manager

The decision was made in Q2 FY25 to migrate to the Director algorithm. Director is considerably more complex than option-critic, and more difficult to implement, but contains several advantages that mitigate the issues seen in option-critic. Most notably, Director, when correctly implemented, has fewer tunable hyperparameters. Many design decisions in the algorithm support a wide range of learning tasks and reward scales, which was thought to be more likely to succeed for the physical security design

problem. Further, the team had experience with model-based RL from prior FY24 work on the adversarial agent project.

4.2. Current algorithm: Director

Director [11] is a hierarchical reinforcement learning algorithm belonging to the Dreamer family of algorithms, which was successfully used to create an adversarial agent in FY24. At a high level, Director is a hierarchical algorithm, that learns from visual representations of the agent's environment. A world model is used to learn representation and planning, a goal autoencoder discretizes goal selection to constrain the manager's action space, a manager selects goals every fixed number of steps to maximize task and exploration rewards, and finally a worker learns primitive actions. Put together, Director consists of about 16 different neural networks [1], compared to option-critic's 4-5. We generally follow Hafner's description of Director in our implementation with a few exceptions. These will be described in the relevant sections of the experimental design (Section 5). However, one significant architectural change was the use of a DreamerV3 [13] [2] recurrent state space model (RSSM) and neural architectures instead of the paper's DreamerV2 architectures. This was done to improve learning stability as DreamerV3 introduced several improvements over DreamerV2; symlog transformations of target values, KL regularization, uniform mix to logits, return regularization, and a few other tricks.

We describe the general flow of the algorithm below, but refer readers to the original papers for Director [11] and DreamerV3 [13] for further implementation details.

Director training flow [3]

1. Replay buffer is initialized and filled. During this stage, actions from the worker are randomly sampled as are the manager goals.
2. The main training loop begins after the replay buffer is filled
3. The current environmental observation is used to create encoded observations. These observations are used with the RSSM representation model along with the prior RSSM state to form the posterior. The RSSM model state is formed through the the concatenation of the recurrent (previous RSSM state) and stochastic (posterior produced by the representation model).
4. Director selects new goals at fixed intervals. If enough steps have elapsed, the manager selects a new goal. The model state is combined with the coordinate embedding to generate a latent goal (compressed dimensionality) before the goal autoencoder inflates the latent goal dimensionality to that of the RSSM latent space.
    a) The coordinate embedding is not part of the standard Director algorithm, but was included to improve learning. More details can be found in Section 5.5.

[1] Specifically, we had 10 different optimizers and 16 different neural networks, depending on how you count

[2] We specifically used the paper's v1 version from 2023 – recent changes described in the paper's v2 version (2024) such as the use of the LaProp optimizer were not implemented here

[3] Inference (evaluation) follows a similar flow, sans the optimization

5. The worker uses the decoded goal and model state along with the coordinate embedding to select actions.
6. Actions are carried out in the environment and a reward is returned.
7. The replay buffer logs the observation, actions, termination state, truncation state, and other properties for later use in the optimization step.
8. The recurrent state of the RSSM is advanced using the actions selected by the worker
9. A crucial hyperparamter controlling model update frequency, called replay ratio, is evaluated. If enough environmental steps have been executed, then a training step is performed
    a) Data is randomly sampled from the sequential replay buffer
    b) Dreamer models learn entirely from imagination. The real data from the replay buffer is used to bootstrap the imagination trajectory.
    c) The RSSM imagines future trajectories entirely in the latent space
    d) The imagined trajectory is used to calculate the world model loss (RSSM, continue model, reward model, observation autoencoder)
    e) The RSSM state is advanced, the goal autoencoder loss is calculated and updated.
    f) The manager and worker losses are calculated. Both extrinsic returns (rewards from the environment) and intrinsic (exploration rewards) are considered.
    g) Manager and worker gradients are clipped, errors are back propagated, and the optimizer steps
10. Loop continues until maximum number of steps are reached

This page intentionally left blank.

5. METHODOLOGY: EXPERIMENTAL DESIGN

We conducted this work using a scaled down version of the problem, in size, but keep the full actions space. Specifically, we consider the design/auto-complete of a physical protection system that's 36x36 pixels in size. This contrasts with a full design which might be about 400x400 pixels or larger. Keeping the initial problem size small lets us more effectively debug the algorithmic implementation and also keeps the compute requirements lower, allowing for quicker iteration. The following sections describe each of the key reinforcement learning components needed to develop an agent. We will focus on the current agent implementation using Director.

5.1. Environment

We begin by adopting a PathTrace-like representation of a physical protection system. The PPS is discretized into cells and used for simulation. Each cell has some semantic meaning and corresponds to a PPS element. For example, one cell could represent a basic wall where another is a detection area. We generally describe performance of a PPS in terms of "path-length". This is a general heuristic that incorporates both the delay of physical elements and probability of detection. We want the designer agent to increase path lengths in a cost effective manner. Generally, path length is a metric that combines detection and delay for a particular path.

The goal of the planner agent is to learn how to make impactful edits to a PPS layout, constrained by user defined metrics, to improve the design from a path analysis perspective. There's two key environmental components for the agent that must be considered; the initial state and the numeric representation of taking an action. These are straightforward for the current state of practice. The initial state is the existing facility layout and components required for site operation (e.g., a turbine hall, reactor containment, etc) and the numeric representation of any edit to the PPS is abstracted by the human brain. For example, a human expert might think "I should build two fences in this area and add a sensor between". The RL agent is being trained from scratch, without any priors, and has no similar abstract understanding of the task or successful strategies. Consequently, a RL agent must have some understanding of the impact of *every* edit before it can reach similar abstract goals.

Since we want an agent that functions as a "improve this area" type of auto-complete agent, we opt to *not* start with a blank slate and instead start with some existing PPS design. This is problematic since an agent needs many steps to learn during training. Generating "valid", randomized, PPS designs is not currently feasible, so we make a rough approximation by generating mazes with similar game rules as those used in path analysis evaluations. There's a wide range of different maze generation algorithms that exist, but simply using these as examples for the RL agent would bias the agent to generate similar solutions (existing algorithms can solve simple mazes). We reduce the impact of this limitation by first training a variational autoencoder (VAE) to generate maze designs. Specifically, we generate maze

layouts from a number of different maze generation algorithms (Aldous-Broder [14, 15], Eller's [16], and Kruskal's [17] among others). Each maze generation algorithm has different strengths and weaknesses, so we use as many as possible.

Although training a VAE doesn't allow us to generate mazes vastly different from the underlying generation algorithms used to create the training data, it does allow us to find interpolated maze designs. For example, sampling the trained VAE latent space can generate mazes that are somewhere in between Eller's and Kruskal's designs. We use the VAE to create maze designs and then use some classical computer vision methods to detect different areas of the the maze and fill different regions with colors corresponding to different PPS sensors and barriers. The output of the maze generation prodecure is a maze-like structure with varying deployments and arrangements of PPS elements.

The maze generation process is done on a "patch" level, each of which is 18x18 pixels in size. We compose patches into a grid to create an entire starting canvas for the designer agent. For example, in this initial work, we consider a 2x2 grid of patches as the total PPS system area (Figure 5-1), but a more realistic size is likely 10x10 patches (Figure 5-2).

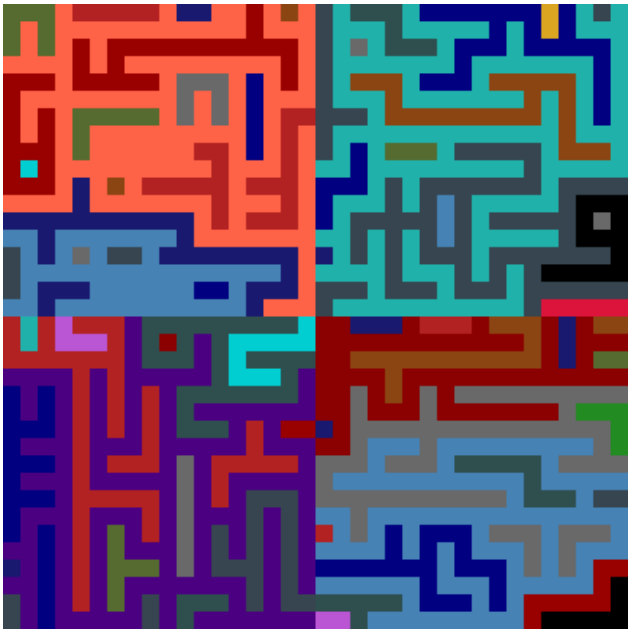


Figure 5-1.: Scaled down maze used in planner agent prototyping. Entire environment is a 2x2 grid of patches.

The goal of the agent is to make high impact edits in these varying, randomized maze designs to increase the difficulty to get to the center [1]. It's more challenging to see the "human-level" solution for these random designs. For example, it's not straightforward to look at Figure 5-1 and know that adding an outer perimeter in the top left patch would significantly improve the design. Consequently, we use a similar paradigm to PathTrace to determine path length and effectiveness to help monitor the agent's overall learning progress. The most vulnerable path, which the planner agent should improve the length of, is determined by a combination of both probability of detection and delay.

[1] This is a simplification for the initial implementation. We recognize that targets will potentially be in different locations depending on facility design

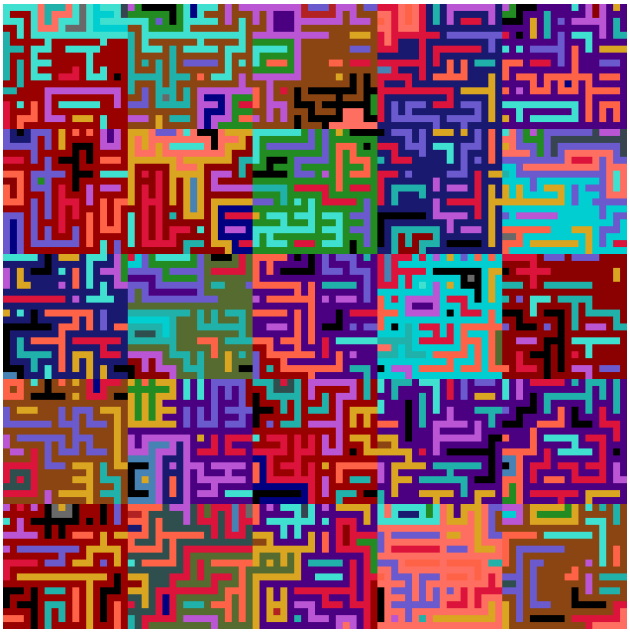


Figure 5-2.: Full scale environment to be used by final agent. Environment is a 10x10 grid of patches.

Two different directed weighted graphs are used to keep track of the path lengths of a maze. The first graph ("probability graph") relies solely on probability of detection for different physical security elements. We assume that the probability of detection for sensors are their highest possible value [2] This assumption was made to develop a proof-of-concept, but are likely more conservative than needed, but they can be easily updated in future work.

The second directed weighted graph ("working graph") keeps track of the actual delays of different elements combined with the probabilities. We assume each PPS element has a delay equal to it's smallest value as a starting point [3]. We use the cumulative probability of detection from each point in the probability graph and multiply it by the actual delay of the protection element at every given cell [4]. This two graph approach can accurately capture the interaction between sensors and barriers. If we assume, for example, you have a very thick wall, but no sensors, then this is still an ineffective configuration as an adversary can slowly bypass the barrier without detection. We use an incremental Dijjstra's algorithm on each [18] graph to calculate path lengths. The incremental algorithms can be very computationally efficient when only making small edits, which will be important when scaling up the planning agent. Both graphs must be updated after every edit is made to the environment.

We treat sensors in a similar way to PathTrace (for simplicity and comparison purposes) in that the probability of detection is checked only once when an area is entered. Any steps within that area after the first check have no impact on detection status. This might reduce learning in the planning agent,

[2]Some sensors perform differently depending on adversary equipment, we assume the highest possible performance

[3]This conservative estimate assumes an adversary would use the most tool with the least delay, at all times, regardless of environmental conditions. We know from prior work this isn't the case, but this assumption is reasonable for a proof-of-concept

[4]We note that we perform a summation of probabilities in a raw sense; if sensors have 99% probability of detection, the graph represents this as 1.98. This isn't statistically valid for an evaluation, but is simple to implement and can be used as an effective learning signal to compare *relative* improvements.

particularly if regions are larger than the imagination horizon, but the detection area probability behavior wasn't considered in depth here.

Another challenge is determining when design is "finished" or "done". Common convention in reinforcement learning is to provide a final end goal state (termination) and/or maximum number of actions in an episode after which to stop the episode (truncation). Both termination (goal state reached) and truncation (maximum number of actions executed) are difficult to formulate in the context of a design process. We only specified a truncation limit and did not specify a termination state. Some limited experimentation was done to consider a termination goal, but no conclusive evidence for a good termination state has emerged yet.

Environment Summary

We used different maze generation algorithms to train a variational autoencoder that can generate maze layouts. We sample the latent space of the autoencoder to generate layouts, which are then have colors applied to represent different PPS elements. Two separate directed weighted graphs are used to calculate the changes in path length due to a single edit. The planner agent is tasked with making edits to these generated designs to increase path lengths, which improves the overall PPS design. We provide an editing budget of 30% of the total cells such that the agent can only modify a subset of all possible tiles.

5.2. Action space

The planner agent worker component is able to take the same actions as a human expert when designing a physical protection system. We adopt the "wide"-like action space representation describe in PCGRL [10]. At each step, the agent selects a tile to modify and which tile to modify it to. Concretely, this would be the agent choosing to modify the tile at $(0, 10)$ to be a basic wall instead of a microwave sensor. We chose to implement this action space as a joint (x, y) space with a separate tile head in order to have maximum design flexibility (in terms of learned solutions) at the cost of a larger action space.

Separate (x, y) action spaces constrain the *shape* of PPS elements "drawn" by the agent. For example, with the independent action space model (i.e., $\pi(x, y|s) = \pi_x(x|s)\pi_y(y|s)$) learning shapes along a diagonal becomes much harder. The two policies could not have a distribution such that $(1, 1), (2, 2,)$ and $(3, 3)$ are likely but that $(1, 3)$, and $(2, 1)$ are not. Additionally, initial exploration noise results in (x, y) combinations that are more shaped like 2D blobs than complex shapes like curves or line strokes.

Both the tile actions and the (x, y) coordinate action use the same learned neural network trunk, but with separate heads. Both the tile and location actions are selected by sampling a parameterized one hot categorical distribution following the typical Dreamer action convention.

Action Space Summary

The worker edits cells in a partial view of the entire design space according to a goal selected by the manager. The worker can select both an (x, y) position of the edit as well as the type of PPS element to place at that position. Specifically, the worker operates in a $18x18$ area at each step and can select one of 22 different PPS elements to place at any cell in the observation area.

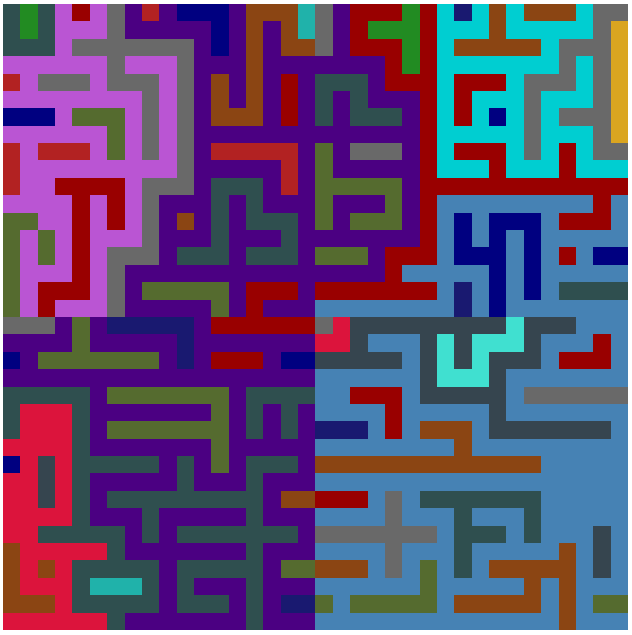


Figure 5-3.: A 2x2 grid of patches used as a starting point for the designer agent.

5.3. Observation space

The observation space refers to the inputs that the algorithm sees and uses to make decisions, which might not include all information from the environment. The original goal was to provide split views to the manager and worker; the manager would receive a global view of the layout whereas the worker would just receive a small section, the size of an individual patch. The reasoning is that by providing the worker a narrow view, it could focus on making local edits whereas the manager would see the entire view and focus on global structure and flow. This works for option-critic wherein the manager and worker can have any arbitrary observation from the environment. However, Director assumes that the manager and worker share the observation space as the manager issues goals in a shared latent space. This means that Director, as originally designed, can't handle split manager and worker views. We currently assume that the worker and manager see the same view of the environment, but future work could expand functionality to handle split views, as can be accomplished in option-critic.

Two different observation schemes were considered. The first was to provide the entire PPS layout as an observation to both the manager and worker. Under this observation scheme, we considered a rotating action space where the manager would choose which patch to edit while the worker retained a consistent action space. For example, the worker could select $(0, 0)$ as an edit cell, but the exact location would vary based on the active patch. In Figure 5-3 below, the $(0, 0)$ could result in an edit in any of the four quadrants, depending on the active patch index.

We recognized this could be confusing to the worker as the same action could lead to n different states (where n is the number of patches in the maze), so we added the current patch index as an observation and as part of the model state used by the manager and worker [5]. We attempted to make the patch index

[5] We intentionally did not include the patch index in the RSSM

a more expressive feature by also passing it through an embedding layer. We found this approach to be unreliable and moved to the second and current approach.

The second observation space approach is partially observable, but more aligned and consistent with how Director expects the worker and manager observations to be aligned. This is a "pan-like" observation where both the manager and worker share the same partial view of the environment. Only a 18x18 pixel subset of the entire layout is passed as an observation [6]. When the worker selects an action the view is then panned. This is inspired by video game playing wherein moving in the environment reveals new part of the area.

Two different "pan-like" movement methods were considered. In the first, the worker must edit a tile in the outer three pixels of the observable space to active the pan, wherein three pixels are panned. In the second approach, any tile except for the center tile pans the view by a single pixel. In both cases, panning does not occur if the pan would be out of bounds of the design area.

Regardless of the actual size of the maze observation, we first bit pack the observations into a discrete categorical representation. This is because each color shown in the mazes has a specific meaning; walls are one color and microwave sensors are another. Using RGB representations of the maze as observations could lead to interpolated RGB values in reconstructions. For example, a reconstructed observation could have an invalid color, something that doesn't correspond to one of the possible PPS elements. Consequently, we bit pack all of the visual maze observations into a categorical representation and use categorical losses for the RSSM's observation loss component. We additionally provide context for where the current view is in relation to the entire maze so that the agent has some understanding of it's relative position within the overall maze.

Observation Space Summary

Both the worker and manager see the same, partial view of the entire design area. The worker can view different parts of the design area by panning the view up, down, left, and right. The worker does this by editing tiles on near the limits of the current view (the worker can also make an edit that doesn't change the selected tile's type and still execute a view pan). Images provided to the worker and manager are represented as discrete integers for each (x, y) position to retain semantic meaning (different colors = different PPS elements). The worker, manager, and RSSM are all provided contextual information on the relative position of the partial view in relation to the larger design area.

5.4. Reward design

Reward design is conceptually straightforward as metrics already exist to evaluate human designed PPS. A good design would balance delay, detection, and cost. The reader might think this a simple task given both delay and probability of detection are recorded and calculated in two directed weighted graphs, however, given the reward sparsity and long actions sequences, reward assignment becomes very

[6] The 18x18 size wasn't chosen for any particular reason other than empirical performance in a few experiments with the autoencoder architecture. This size could be changed in the future, although it would also require expanding the action space size.

challenging. For example, consider drawing a rectangle wall around a target. How could a reward be effectively assigned to each of the edits, during run time? A sequence of three horizontal wall edits could end up being something totally different than a rectangular perimeter, and on their own, they have little value (an adversary can simply walk around the wall).

We formulate three potential-based reward functions that are combined and provided to the agent at each step. Potential-based functions were specifically selected as they result in invariant policies [19] and we don't want the planner to learn effective solutions solely based on Dijkstra calculation [7]. However, simply calculating the shortest path across the entire PPS, which we want to improve, can result in a very sparse reward. Consequently, we consider three different rewards, each of which exist on a different scale. Full details can be found in Appendix B.

- **Global:** This reward component captures the global structure of the entire PPS. This is a sparse reward that encourages high impact edits. This reward is further scaled by the minimum path length across the entire maze to encourage edits in high impact areas.
  - Path lengths are calculated across valid adversarial starts. This includes the outer perimeter border cells only.
- **Patch:** This reward is denser the global reward and captures improvements in the current observation window. The patch reward encourages improvements in local structure.
- **Local:** The final reward component is the most dense and guarantees some reward for every edit. Here, the path length is evaluated from each edited cell to the goal cell.
- **Final:** We provide a final reward on episode end. This is identical to the global reward in form, but is differenced over the entire episode length instead of a single step.

These rewards are penalized with a cost term representing the estimated financial cost of an element. For example, a wooden door is cheaper than a steel reinforced door. The actual estimated financial costs of an element can vary over a wide range, so the costs are scaled by the 95th percentile of the maximum cost. A hyperparameter, ϵ is used to control the impact of cost on the action (e.g. higher ϵ means a more cost sensitive policy).

The scale of the global, patch, and local can vary dramatically, so the values are scaled using a power law transformation. The scaling helps prevent one reward term dominating the others and allows us to control relative importance (in work so far, we apply equal weight to each term). Further, since Director performs a symlog transformation on the rewards, by default, it expects rewards in the $[-20, 20]$ and not much larger. We use the power law as follows:

First define the normalization constant, k, as:

$$k = \frac{T}{R_{\max}(R_{\max} + \epsilon)^{p-1}} \tag{5.1}$$

[7] If we provided the algorithm a raw reward, over time, the agent could just learn to be an inverse-Dijkstra's calculator of sorts and overfit to our training setup. The idea agent will develop novel strategies that might not well represented in the randomized initial mazes or revealed by the Dijkstra's shortest path.

Where T is the target value scale, $R_{\max}$ is the maximum existing value, ϵ is the expected average value.

Then values (r_0) can be scaled as:

$$g(r_0) = kr_0(|r_0| + \epsilon)^{p-1} \tag{5.2}$$

Reward Design Summary

Developing a dense reward system that shapes complex PPS design is challenging. We develop rewards at three densities and temporal scales. Global, patch, and local rewards are implemented, which capture improvements on different scales. Director only requires one explicit reward specification, which is used directly by the Manager and combined with an intrinsic exploration reward. The worker needs no explicit reward as it's reward is based on how well it is following the Manager's chosen goal.

5.5. Algorithmic changes

We started with the reference implementation of Director, but made several modifications to improve performance for the PPS design application. We provide a brief list here:

- **DreamerV3 RSSM:** The original Director paper was based on DreamerV2, but in our implementation, we update the algorithm to utilize advances from DreamerV3. This includes using symlog transforms on predicted values, KL loss regularization, KL loss uniform mixing, return normalization, twohot distributions for critic targets, and the use of SiLU activations with LayerNorm on all MLP networks. We updated the RSSM and all MLP networks (actors, critics, goals, and continues) to use the updated DreamerV3 conventions. We use the XL DreamerV3 model configuration. Further details can be found in Appendix A.

- **Multiple Worker Heads:** We extend the worker's actor architecture to support selection of multiple actions, namely (x, y) coordinates in the editing area and a n value that represents the type of tile to place. We also adjust the entropy coefficient to compensate for having multiple action heads.

- **Coordinate view embeddings:** The current environmental implementation uses a "pan-like" movement system wherein edits to the currently viewed area casues the view to pan. Although benchmarks of DreamerV3 on partially observable problems like Crafter [20] using only a pixel observation space are promising, providing contextual observations can help accelerate learning. We gather the center coordinates of the current view area and pass them through a learning embedding layer to provide context for the worker, manager, and RSSM. The embedded coordinates are injected at different places depending on the specific component and are thought to improve learning. Future work will perform an ablation study to more closely consider the impact of these embedded coordinates on overall learning and solution quality.

- **Attention-based observation autoencoder:** This prototype planner agent operates on a relatively small observation of a 36x36 pixel area. While small, each pixel requires attention to detail due to the underlying semantic meaning. Simple CNN architectures, such as the commonly used Nature CNN architecture [21], are generally not sufficient to recreate fine details in images needed for this problem. Consequently, we developed an attention-based CNN autoencoder that can better resolve pixel-level features. The encoder is flexible, with adaptive pooling, to potentially support split manager/worker views in the future. The two-layer encoder encodes the categorical observation combined with the embedded coordinate feature into the latent space. The corresponding two-layer decoder focuses on only reconstructing the categorical observation as the embedded coordinate feature is provided only for context. The autoencoder operates on the bitpacked observations, which are passed through an embedding layer for richer representation, before being passed through the main trunk.

A more complete list of hyperparameter and component configurations can be found in Appendix A.

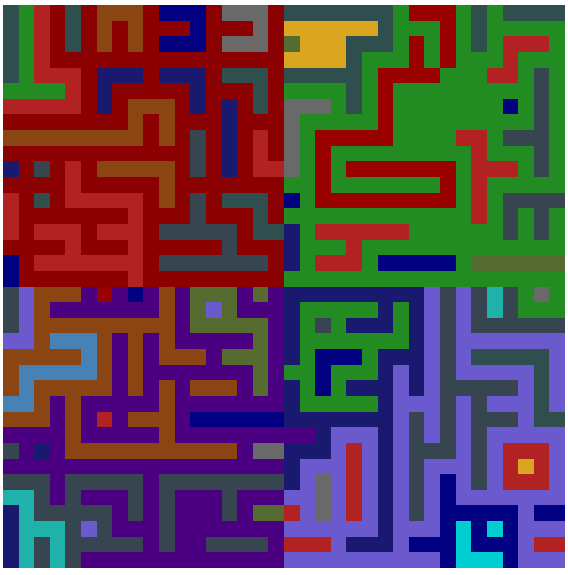
This page intentionally left blank.

6. EXPERIMENTAL RESULTS

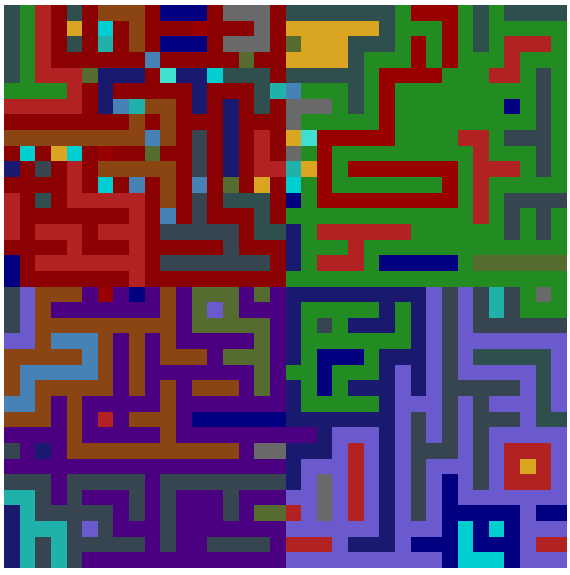
This section presents findings collected at the conclusion of FY25 work. Given the the low TRL nature of this work, and the initial challenges with algorithmic stability (swapping from Option-Critic to Director), these results are going to be less focused on strict quantitative benchmarks and more about trends and qualitative behaviors. We are planning on working with stakeholders to assess interest for FY26 and beyond. If there's significant interest, this work will be polished and future reports will provide more concrete results.

6.1. Initial findings

We were able to develop the necessary algorithmic changes and environmental specifications to ensure that an agent could learn to perform design of a randomized environment. In this work, we have demonstrated that a HRL algorithm can be used for design-like problems, albeit significant work remains to transition this concept to a production ready tool. Some initial results from the algorithm are shown below in Figure 6-1, with images for before and after edits have been made.



(a) Randomly generated layout before edits have been made.



(b) Randomly generated layout after edits have been made.

Figure 6-1.: Maze after agents has made edits. Note that edits are concentrated in the top left patch, whereas the others have been unedited.

The edits made by the agent are interesting and do generally increase the path length in that patch, however, it's obvious that the agent is only making edits in a small region and is not editing the entire

maze. In fact, the agent makes it's most impactful edit in the first 10% of the steps taken, then proceeds to undo/redo the same edit for the rest of the episode length.

First, it's believed that the edit concentration being in a particular path is due to some combination of the limited observation view, dense local reward, and limited panning behavior (by default):

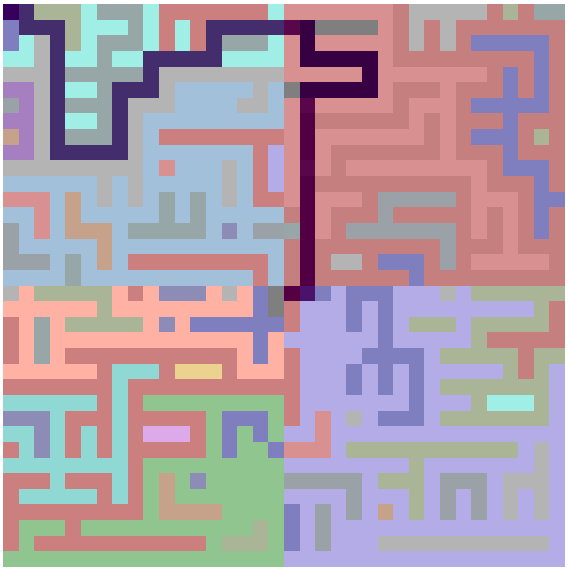
- **Limited observation:** The agent only sees a partial view of the maze at once, primarily to constrain the size of the action space. Selecting (x, y) coordinates over an 18x18 view is going to be much easier than over a 36x36. Related work, particularly with the Crafter environment [20], has shown that Dreamer algorithms (including Director) can effectively learn to act in partially observable environments, so it's unlikely this is the root cause.
- **Dense local rewards:** We provide a three part reward to the agent. The local component, which measures the effective change along the path of the edited cell, is the most dense. The other components are more sparse and perhaps less preferable, at least in early learning.
- **Limited panning behavior:** Our baseline environment lets the agent pan the view around the maze by making edits around the edges of the current view. If the agent doesn't learn an explicit "move" behavior, it will likely sit in the same location. A "move" action might be simply placing a PPS element of the same kind on an edge cell, which would result in a zero reward, but pan the view. It's unclear if the intrinsic reward component of Director can uncover this behavior or if some reward shaping is required.

It's likely that the dense local reward combined with the panning behavior together work to reduce the agent's panning to other regions of the design area. The frequent local rewards does little to encourage panning behavior (even if the less dense patch, global, and episode rewards do). The chances that panning is done by "accident" during early training. We performed some initial exploration into modified panning behavior, however, modifying local reward density has yet to be explored.

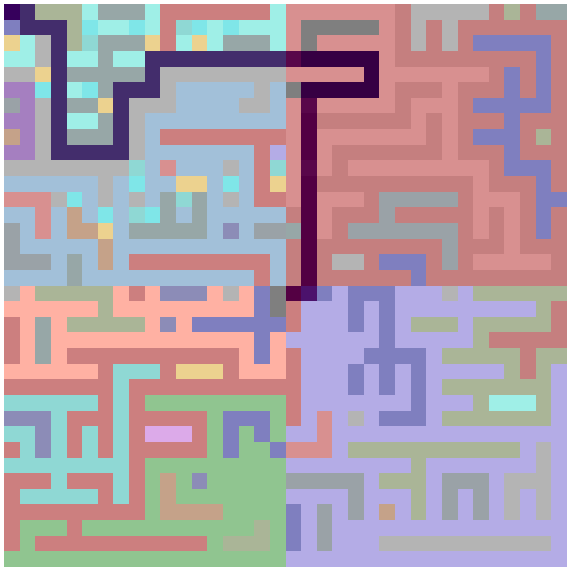
Considering that an agent randomly selecting an outer edge of it's local view for an edit is rare (especially when actions are nearly random during early training), we ran a few experiments at different panning strategies. We primarily considered an "all-pan" strategy where *all* edits caused a pan of a single pixel, with the pan direction being dependent on the edit location. We didn't observe successful learning in these experiments, likely due to the sheer number of novel states generated through constant panning. We believe that the "all-pan" strategy didn't work due to the agent receiving strong early intrinsic rewards, which encouraged it to improve those. More details on the extrinsic/intrinsic reward trade-off is discussed in Section 6.2.

Beyond the limited scope of edits made by the agent, the agent tends to make the most impactful edits early in the episode then undo/redo edits, which averages out to zero rewards. This behavior is a bit more unclear, but perhaps the lack of a episode termination state along with the agent's static behavior result in an agent's "done" behavior being redundant actions.

Although the agent's panning ability needs improvement, the edits it does make are generally useful and do meaningfully increase the path length. The shortest path for a given layout, before and after edits, is shown in Figure 6-2 below. Note the change in the path shape itself, but the path cost changes significantly even though the number of cells doesn't change dramatically.



(a) Randomly generated layout before edits have been made.



(b) Randomly generated layout after edits have been made. Note that both the path itself has changed as has the cost of the path.

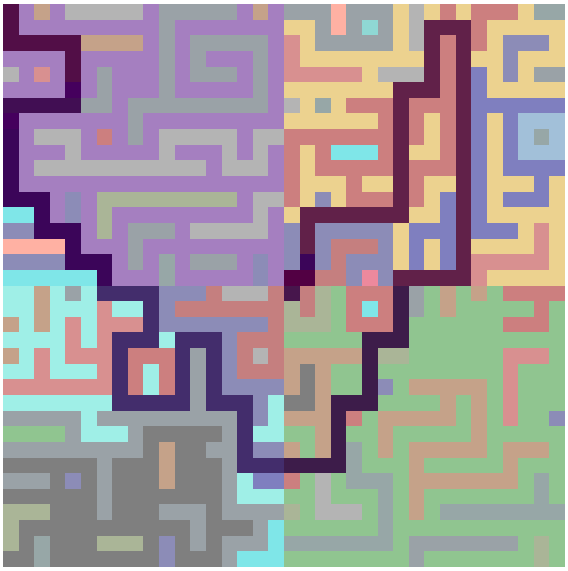
Figure 6-2.: Maze after agents has made edits. Note that edits are concentrated in the top left patch, whereas the others have been unedited. The highlighted path length (shortest path from a fixed origin in the top left) has changed, as has the entire global path lengths and top left patch path length.

However, results aren't necessarily always consistent. For example, in Figure 6-3 below, the path length for a given start dramatically decreases after the agent makes edits. We think this is likely due to the agent's limited view; the agent doesn't sufficiently consider patches outside of it's view (or the view changes very little).

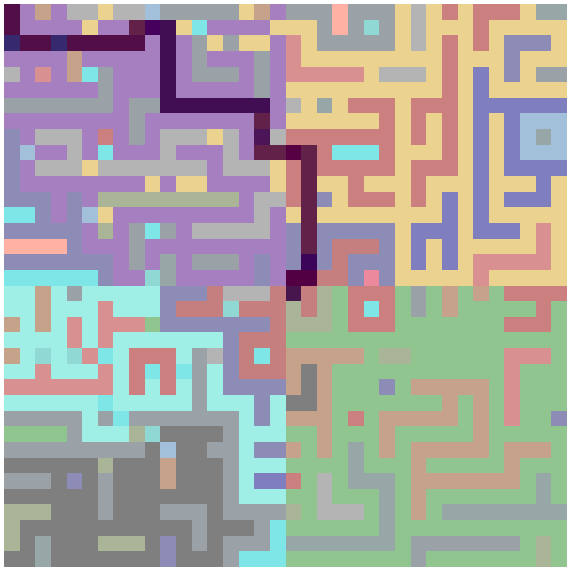
> **Initial findings summary**
>
> We have found that the planning agent can learn effective edit sequences to improve randomly generated PPS layouts, but it's inconsistent from episode to episode and tends to only edit one part of the map. We think that that is likely due to the a combination of the action design (hard to randomly move the agent's view), dense local rewards (lots of rewards for edits in the starting location, so why pan?), and challenges with intrinsic rewards (more on that below).

6.2. Training stability

Reinforcement learning can be sensitive to initial state (e.g., weight initialization, first layouts seen, and the initial sequences of random actions), which is why results are often aggregated over multiple randomized starting conditions. We have found that performance can vary wildly across different starts. Some runs will start in bad initial positions and never recover; consistently obtaining negative rewards and never improving. Yet others will find early initial success and exploit strategies extended lengths of time. Others might find successful strategies, then collapse part way through training.



(a) Randomly generated layout before edits have been made.



(b) Randomly generated layout after edits have been made. Note the path length (and cost itself) has decreased significantly, which is a worse result.

Figure 6-3.: Maze after agents has made edits. The new path length is significantly worse than the initial state.

We think that intrinsic rewards are a major driver for instability. The actual actions taken by the manager and worker are driven by the λ-returns:

$$\begin{aligned} R_t^\lambda &= r_t + \gamma c_t \left((1-\lambda) v_\psi(s_{t+1}) + \lambda R_{t+1}^\lambda \right) \\ R_T^\lambda &\approx v_\psi(s_T) \end{aligned} \tag{6.1}$$

Where the critic is is v_ψ. It's important to note that while the worker has a single return (intrinsic reward based on following manager goal objective), the manager has *two* returns, both extrinsic and intrinsic. The extrinsic returns are derived from our specifications in the environment (i.e., the reward we specify), but the intrinsic is based on the autoencoder reconstruction error. The idea here is that model states that are reconstructed poorly are "novel" and should be visited by the agent. This encourages exploration of new states and, in theory, helps avoid narrow and specific strategies that don't generalize to many states.

The manager uses both terms to calculate the actor's advantage, which informs how the agent is trained. Director specifies a baseline intrinsic advantage scalar of 0.10. That is, the final advantage term is 0.90 extrinsic and 0.10 intrinsic. We have observed that this value might be too high and contribute to several of the problem we've seen. Runs that exhibit particularly poor performance start with large negative extrinsic returns as the agent searches the solution space for good strategies. During this time, the intrinsic reward can be quite large as the world model is still learning. The agent can learn early that producing new states (and thus maximizing intrinsic rewards) is easier than finding good strategies to maximize the actual design objective. The agent then solely focuses on maximizing intrinsic reward and never progresses. The 0.10 factor is not a concrete rule and is a high priority for investigation for future work.

7. CONCLUSION

This work demonstrated one of the first uses of machine learning for a design problem in the nuclear energy space. Although this work is far from complete, we demonstrated a working prototype with concrete ideas and a path forward. We identified an algorithm that can learn successful strategies for PPS design along with several likely areas for improvement. These include the following:

- **Intrinsic advantage multiplier:** Director uses both an intrinsic and extrinsic reward by default. Under certain conditions, the agent might learn early to maximize intrinsic reward since it's easier to accomplish, particularly in a randomized environment, then learning viable strategies to improve extrinsic reward. We will consider reducing or removing the manager's intrinsic reward in an effort to improve training stability.
- **Model size:** We primarily looked at the XL Dreamer model size [1], which controls the depth and number of dense units in the trainable neural networks. We've performed some limited experimentation that shows a smaller model, perhaps as small as the M size might work similar or better with improved training speed.
- **Pan action behavior:** We found that the panning action view is an improvement over the previously considered patch rotation strategy. This behavior better aligns the observation space of the manager and worker (as opposed to the worker learning correlations between action location and patch index). However, the agent has a tendency frequently sit in its' starting position. This is likely due to poor sampling at the outer edges that trigger the pan. We will look at different view shifting strategies to promote better coverage over the observation space.
- **Extended training regimes:** This work focused on rapid iteration given the level of technical challenge presented. The current implementation has limited performance in that it can only use a single GPU at once. This puts a ceiling on how much training can be done for any given experiment. Prior work in FY24, for the adversarial agent, took as many as 200M environmental steps of training (32 GPU days). This problem is much smaller in scale than the adversarial work, but would likely benefit from longer training periods.

Long term, we hope to develop an integrated design stack combining both the adversarial agent and planner agent together, through self play, to realize AI-driven design. Prior work in FY24 showed that a model-based RL agent could successfully learn to exploit vulnerable facility paths. The adversarial agent could ultimately be trained concurrently with the planner agent. Instead of using Dijkstra's algorithm to calculate path-based rewards, the planner could improve iteratively with the planner, learning from unique strategies uncovered by self-play.

[1] This size was successful in prior FY24 work wherein anything smaller than L failed to train properly. Although the environments are different, that context provided a starting point for our model size selection

This page intentionally left blank.

REFERENCES

[1] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[2] A. S. Evans, "Small modular reactor and microreactor security-by-design lessons learned: Integrated pps designs," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2024.

[3] M. L. Garcia, *Design and evaluation of physical protection systems*. Elsevier, 2007.

[4] A. T. Orr, P. W. Zahnle, and J. Miller, "Pathtrace and mpveasi: A path analysis comparative validation study," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2023.

[5] W. Wong, P. Dutta, O. Voicu, Y. Chervonyi, C. Paduraru, and J. Luo, "Optimizing industrial hvac systems with hierarchical reinforcement learning," *ArXiv*, vol. abs/2209.08112, 2022.

[6] G. Kwon, B. Kim, and N. K. Kwon, "Reinforcement learning with task decomposition and task-specific reward system for automation of high-level tasks," *Biomimetics*, vol. 9, 2024.

[7] W. Lu, Y. Wei, J. Xu, W. Jia, L. Li, R. Xiong, and Y. Wang, "Reinforcement learning for adaptive planner parameter tuning: A perspective on hierarchical architecture," *ArXiv*, vol. abs/2503.18366, 2025.

[8] J. Bae, J. M. Kim, and S. J. Lee, "Deep reinforcement learning for a multi-objective operation in a nuclear power plant," *Nuclear Engineering and Technology*, vol. 55, no. 9, pp. 3277–3290, 2023.

[9] J. Park, T. Kim, S. Seong, and S. Koo, "Control automation in the heat-up mode of a nuclear power plant using reinforcement learning," *Progress in Nuclear Energy*, vol. 145, p. 104107, 2022.

[10] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, pp. 95–101, 2020.

[11] D. Hafner, K.-H. Lee, I. Fischer, and P. Abbeel, "Deep hierarchical planning from pixels," *Advances in Neural Information Processing Systems*, vol. 35, pp. 26091–26104, 2022.

[12] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. Van Hasselt, "Multi-task deep reinforcement learning with popart," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3796–3803, 2019.

[13] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[14] D. J. Aldous, "The random walk construction of uniform spanning trees and uniform labelled trees," *SIAM Journal on Discrete Mathematics*, vol. 3, no. 4, pp. 450–465, 1990.

[15] A. Z. Broder, "Generating random spanning trees," in *FOCS*, vol. 89, pp. 442–447, 1989.

[16] J. Buck, "Mazes for programmers: Code your own twisty little passages," 2015.

[17] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[18] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: his life, work, and legacy*, pp. 287–290, 2022.

[19] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Icml*, vol. 99, pp. 278–287, Citeseer, 1999.

[20] D. Hafner, "Benchmarking the spectrum of agent capabilities," *arXiv preprint arXiv:2109.06780*, 2021.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

APPENDIX A. Algorithm Hyperparameters

Hyperparameters used in the Director implementation are listed below. Parameters for the DreamerV3-based RSSM follow the XL configuration.

Name	Symbol	Value
General		
Replay Capacity (FIFO)	-	10^6
Batch Size	B	16
Batch Length	T	64
Activation	-	LayerNorm + SiLU
Parallel Envs	-	32
Replay Ratio	-	0.015625
MLP Size	-	$5x1024$
World Model		
Stochastic State Size	h	4096
Deterministic State Size	z	1024
Obs Encoder Latent Size	-	1024
Number of Latents	-	32
Classes per Latent	-	32
Reconstruction Loss Scale	β_{rep}	0.1
Dynamics Loss Scale	β_{dyn}	0.5
Representation Loss Scale	β_{pred}	1.0
Learning Rate	-	10^{-4}
Adam Epsilon	ϵ_{WM}	10^{-8}
Gradient Clipping	-	1000
Goal Autoencoder Latents	L	8
Goal Autoencoder Classes	C	8
Goal Autoencoder Beta	B	1.0
Goal Duration	K	8
Actor Critic		
Imagination Horizon	H	15
Discount Horizon	$1/(1-\gamma)$	333
Critic EMA Mix	α	0.001
Return Normalization Scale	S	$\text{Per}(R, 95)$ - $\text{Per}(R, 5)$
Return Normalization Limit	L	1
Return Normalization Decay	-	0.99
Actor Entropy Scale (Manager)	-	10^{-3}
Actor Entropy Scale (Worker)	-	10^{-3}
Learning Rate (Manager)	-	$8 * 10^{-5}$
Learning Rate (Worker)	-	$8 * 10^{-5}$
Adam Epsilon	ϵ_{AC}	10^{-5}
Gradient Clipping	-	100

APPENDIX B. Reward functions

The following sections discuss the different reward components used for the planning agent. We usually assume these components are weighted equally, but more work is needed to determine if there is an optimal balance.

B.1. Global

Assume we have some graph $G = (V, E)$ then we can calculate the average path length change in a potential form as follows:

$$\left(\frac{\sum_{s \in G} \text{dist}_G(s, v)}{|S|}\right)_t - \left(\frac{\sum_{s \in G} \text{dist}_G(s, v)}{|S|}\right)_{t-1} \tag{B.1}$$

In practice, dist_G is calculated from the second working graph, which represents physical protection system delays and probabilities of detection. Since we also want to constrain by other metrics instead of optimizing for sheer performance, we additionally constrain by cost.

$$\left(\frac{\sum_{s \in G} \text{dist}_G(s, v)}{|S|}\right)_t - \left(\frac{\sum_{s \in G} \text{dist}_G(s, v)}{|S|}\right)_{t-1} - \epsilon * (\Delta \text{ cost}) \tag{B.2}$$

Where epsilon is some hyperparameter controlling the performance/cost tradeoff and Δ cost can simply be calculated by summing the value of all cells in the graph.

B.2. Patch

The patch reward is identical in form to the global reward with a different scope. The patch reward looks at valid starts for the agent's current view instead of the entire list of possible adversary starts. If the patch is an interior patch (that is, none of the patch's borders are shared with the global map border), then all possible periphery cells are valid starts. However, if some of the patch's border overlaps with the global map border, then we assume the interior border is not a valid start. For example, in a 3x3 grid, we would assume that the top middle cell would not have a valid start position at it's southern border.

B.3. Local

The local reward is similar to the patch reward, but with the notable difference that the local reward only has a single possible origin point, the edited cell. Instead of calculating the average path length across multiple starts, the local reward assigns credit based on the distance from the edited cell to the goal.

B.4. Episode Truncation

A final reward is provided when the episode has been truncated. This is identical to the global reward, but the key difference is that the episode reward looks at the total change over the episode, not just a single step.